1994

Generational Garbage Collection of C++ Targeted to SPARC Architectures

Satish K. Guggilla
Iowa State University

Generational Garbage Collection of C++ Targeted to SPARC Architectures

Abstract
Dynamic memory management plays a crucial role in the development of large software systems. Traditional techniques for managing dynamic memory require the programmer to free an allocated object when it is no longer required. In addition to posing an intellectual burden on the programmer, this approach has often proved error-prone. Many bugs in existing software systems are known to be caused by dynamic memory management errors. Garbage collectors free the programmer from this intellectual burden by automatically reclaiming allocated objects that are no longer in use. In systems with garbage collection, the programmer need not concern himself with releasing objects no longer in use. Most traditional garbage collectors suspend the application program during the collection process. Generational garbage collectors are known to achieve short pause times as they rely on the observation that most objects die young. They concentrate most of their efforts in reclaiming recently allocated objects, occasionally performing a complete collection. In this project, we have implemented a generational garbage collector for C++ targeted to SPARC architectures. Our technique imposes only minor restrictions on the usage of dynamic memory in C++ and runs on stock hardware. * Portions of this paper were excerpted from "Code Generation to Support Efficient Accurate Garbage Collection of C++ on Stock Hardware", a paper currently being prepared for publication by Kelvin Nilsen, Ravichandran Ganesan, Satish Guggilla, Satish Kumar, and Kannan Narasimhan.

Disciplines
Systems Architecture

Generational Garbage Collection of C++ Targeted to SPARC Architectures

Satish Kumar Guggilla[1]

Department of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50011

ABSTRACT

Dynamic memory management plays a crucial role in the development of large software systems. Traditional techniques for managing dynamic memory require the programmer to free an allocated object when it is no longer required. In addition to posing an intellectual burden on the programmer, this approach has often proved error-prone. Many bugs in existing software systems are known to be caused by dynamic memory management errors. Garbage collectors free the programmer from this intellectual burden by automatically reclaiming allocated objects that are no longer in use. In systems with garbage collection, the programmer need not concern himself with releasing objects no longer in use. Most traditional garbage collectors suspend the application program during the collection process. Generational garbage collectors are known to achieve short pause times as they rely on the observation that most objects die young. They concentrate most of their efforts in reclaiming recently allocated objects, occasionally performing a complete collection. In this project, we have implemented a generational garbage collector for C++ targeted to SPARC architectures. Our technique imposes only minor restrictions on the usage of dynamic memory in C++ and runs on stock hardware.

Introduction

Dynamic management of memory allows memory to be used for different purposes during different execution phases of a program. Without dynamic memory management, application developers are often forced to statically allocate memory for all the anticipated needs of their applications. This results in artificially rigid constraints on applications and leads to poor utilization of memory. Buffers are not allowed to expand or shrink depending on changing circumstances. Static allocation results in increased hardware costs, because each word of memory serves only one purpose throughout the execution of the application. Segments of memory that are not currently in use sit idle. They cannot be temporarily reallocated to serve different needs. Dynamic memory systems allow sharing of memory between different execution phases of an application to reduce system costs. They allow common memory resources to be shared between multiple components and facilitate flexible system reconfiguration in response to changes in workload or operating environment. Dynamic resizing of buffers and other data structures is possible depending on the current application requirements so as not to arbitrarily restrict user interaction and data inputs.

In a system with dynamic memory management, a user process is often provided with a free pool of memory, referred to as the *heap*. All the allocation requests for dynamic memory are satisfied from the heap. A user process allocates dynamic memory with the new operator, which is implemented using the malloc function. Traditional techniques for dynamic memory management maintain linked lists of free memory segments. These are called *free lists*. Allocation involves traversing the free lists in search of a segment of a size that is appropriate to satisfy the allocation request. The application developer uses the delete operator, which is implemented in terms of free, to return previously allocated memory to the free pool. free links the released object onto the free list, possibly coalescing it with neighboring free objects before linking it onto the appropriate list.

[1] Portions of this paper were excerpted from *Code Generation to Support Efficient Accurate Garbage Collection of C++ on Stock Hardware*, a paper currently being prepared for publication by Kelvin Nilsen, Ravichandran Ganesan, Satish Guggilla, Satish Kumar, and Kannan Narasimhan.

The term *garbage collection* describes the automated process of finding previously allocated memory that is no longer in use in order to make the memory available to satisfy future allocation requests. An object is considered *garbage* if it is not reachable by the running program via any path of pointer traversals. An object that is reachable via some path through a pointer traversal is termed as *live* and is preserved by the collector. The basic functioning of a garbage collector consists of two parts:

1. Distinguishing the live objects from the garbage (garbage detection).
2. Reclaiming the memory occupied by garbage (garbage reclamation).

To determine the liveness of an object, a garbage collector typically uses a simple criterion, defined in terms of a *root set* and *reachability* from these roots. The root set contains all globally visible variables of active procedures, the local variables in the activation stack and the registers used by the active procedures. Heap objects directly reachable from this root set are accessible by the running program and must be preserved. In addition, since the program might traverse pointers from these heap objects to reach other objects, any object reachable from a live object is also live. The set of live objects, therefore, is simply the set of objects on any directed path of pointers starting from the root set.

Any object that is not reachable from the root set is garbage. No 'legal' sequence of program actions would allow the program to reach this object and the memory occupied by this object can thus be safely reclaimed.

Motivation for Garbage Collection

Garbage collection is necessary for fully modular programming. In traditional programming languages, programmers are required to explicitly free memory that is no longer required. For modular programming, a routine operating on a data object should not have to know what other routines may be operating on the same object unless dictated by the application needs. If explicit deallocation is required, some module must be responsible for knowing when other modules are not interested in a particular object. Since liveness is a global property, this will involve nonlocal bookkeeping in modules that might otherwise be orthogonal and reusable. This nonlocal bookkeeping reduces extensibility, as an addition of a new module which operates on the same object will require that the bookkeeping code be updated.

Also, extreme caution must be exercised in freeing an object at the correct time. Holding on to an object for too long or releasing it too early both lead to severe problems. Failure to reclaim memory at the proper point of time may lead to slow memory leaks which may eventually exhaust the free pool. On the other hand, reclaiming memory too soon can lead to very strange behavior. If an object is accidentally returned to the free pool and then reallocated while it is still serving the purpose for which it was originally allocated, then the same memory serves two purposes. Since each of the users of the shared memory segment thinks that it is the only user, the users will become confused when the values stored into the memory segment by one of the users are overwritten by one of the other users. Such errors are often very hard to debug because the consequence of an error is generally not detected at the time the error occurs. In some cases, the error might go unnoticed until after release of a commercial product since the symptoms of the error are only manifest when the software system is stressed in certain ways.

These problems often lead many programmers to develop application specific garbage collection when faced with the task of developing a large software system. However, these collectors are often incomplete and contain errors as they are coded for a one-time application. They are often unreliable, in addition to being hard to reuse as they are not integrated into the programming languages.

Automatic garbage collection greatly simplifies the development effort required to manage dynamic memory. In systems that provide garbage collection, programmers need not be concerned about explicitly freeing memory that is no longer in use. Besides reducing the programmers intellectual burden, automatic garbage collection eliminates a variety of common programming errors. Automatic garbage collection offers the potential of reducing the costs of developing large software systems by approximately 40% of the costs required to develop the same software without automatic garbage collection [11]. Besides reducing the complexity of dynamic memory management, some modern garbage collection algorithms offer storage throughputs that are often better or comparable to traditional heap management techniques.

Motivation for the Current Project

Traditional garbage collector implementations periodically suspend application processing in order to traverse all of memory in search of segments that are no longer in use. Garbage collection time is proportional to the number of

live objects. In a large number of programs written in a variety of languages, it has been observed that most dynamically allocated objects live only a short time. Usually between 80 and 98 percent of all newly-allocated objects die within a few million instructions, or before another megabyte has been allocated; the majority of objects die even more quickly [15]. The pause times observed during garbage collection are proportional to the amount of live data and the size of the heap. Generational garbage collectors attempt to reduce pause times by concentrating on reclaiming recently allocated objects. They make use of the observation that most objects die young by treating objects differently depending on how long they have survived. These garbage collectors ignore long-lived objects, the *stable set*, and concentrate on reclaiming space occupied by short lived objects. They avoid scanning and copying long-lived objects. In the average case, they thus perform better than traditional garbage collectors. In the worst case, their performance is only slightly worse than traditional garbage collectors.

As part of this project, we have implemented a generational garbage collector for C++ targeted to SPARC architectures. We have also been motivated by the fact that existing garbage collection techniques for C++ often impose unnecessary and undesirable restrictions on the language which impede acceptance of garbage collection techniques by the C++ community. Our technique does not impose any unnecessary restrictions on the language. The usefulness of our garbage collection technique has been demonstrated earlier in references 11 and 13. Work is in progress to develop the prototype hardware described in references 11 and 13. Our generational collector, however, is designed to run on stock hardware and does not require any assistance from special-purpose hardware. We would like to have software-supported garbage collection that is compatible with our hardware-collected C++ dialect as this will encourage standardization of our technique. This will also enable us to conduct further compiler studies to prove the efficacy and help in the acceptance of our technique by the C++ community. This project aims to be a stepping stone in the achievement of this goal.

A group of three graduate students is working on making the necessary changes to GNU's C++ compiler for supporting garbage collection. These changes are described in detail in the later sections. The remainder of this document is organized as follows. In section 1, we briefly review some of the available garbage collection techniques including generational garbage collection. Section 2 describes the changes being done to the C++ compiler. Section 3 discusses allocation of heap objects. We describe the garbage collection algorithm in section 4. Section 5 concludes with our observations and suggestions for future work.

1. Garbage Collection Techniques

In this section, we briefly review some of the popular garbage collection techniques. Our review follows the discussion provided in reference 15. This section makes the simplifying assumption that heap objects are *self-identifying*. What this means is that it is easy to determine the type of an object at run time. This is essential to locate pointers within the object and to determine the object's size if it needs to be relocated during reclamation. Heap allocated objects typically have a header field which stores information such as size of the object and an encoding of its type. The header is normally hidden from the application program. Section 3 provides more details on how this can be implemented. We refer the reader to reference 15 for a more detailed survey of available garbage collection techniques.

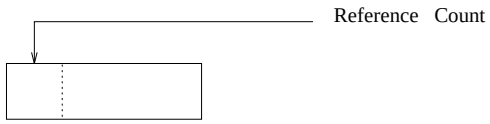
Reference Counting

In this garbage collection technique, a *count* field is associated with each heap allocated object. This field is often referred to as the *reference count*, and is used to maintain the number of references to the corresponding heap allocated object. Whenever an assignment is made to a location known to contain a pointer, the following actions are taken:

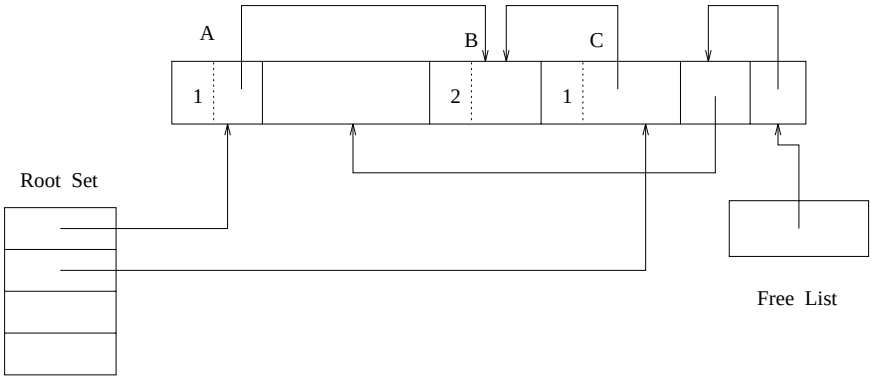
1. The reference count of the object pointed to by the pointer location prior to the assignment is decremented by 1.
2. The reference count of the object pointed to by the pointer location after the assignment is incremented by 1.

Allocation involves searching through a free list of objects until an object of appropriate size is found. Allocation fails if no such object is found. In this method, an allocated object becomes garbage once its reference count becomes zero. Reclamation involves linking garbage onto the free list. The reference counts of the objects pointed to by a reclaimed object are automatically decremented. Reclaiming a single object may therefore cause many other objects to be reclaimed.

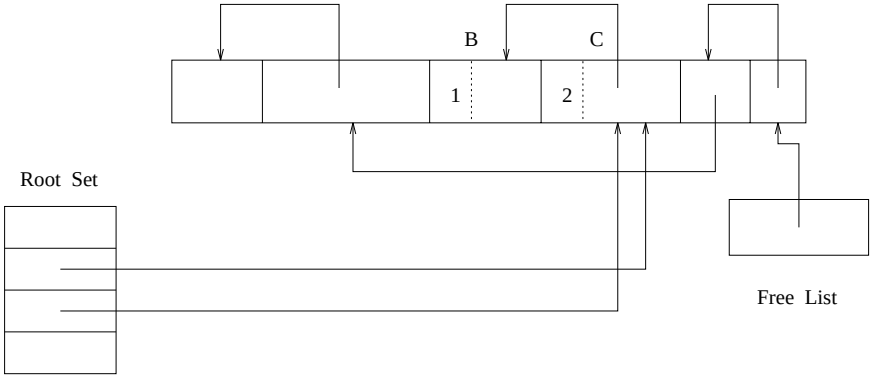
This technique is illustrated in Fig. 1.1. As can be seen from Fig. 1.1.A, a heap allocated object has a field to maintain the reference count. Fig. 1.1.B gives the layout of the heap at some point of time. The organization of the same heap is shown in Fig. 1.1.C after the deletion of the pointer to object A and creation of a new pointer to object C. Object A becomes garbage as its reference count falls to zero. Consequently, it is linked onto the free list. The reference count of B gets decremented automatically as A points to it. The reference count of C increases by 1 due to the creation of a new pointer to C.



A) Format of a heap allocated object



B) Heap organization at some instant of time



C) Heap after deletion of pointer to A and creation of another pointer to C

Fig. 1.1: Reference Counting Collectors

The main advantages of this technique are the simplicity of implementation and its incremental nature. Most of the garbage collection work is interleaved with actual program execution. It is easy to make these collectors real-time ensuring that at most a bounded amount of garbage collection is done per unit of program execution [15][2].

[2] Note that compliance with real-time constraints requires deferred reference counting, in which objects are placed onto the free list without decrementing the reference counts of the objects referenced by the newly freed object. When an object is reallocated from a free list, all of the pointers contained within the object are set to NULL, and the reference counts of the objects previously referenced by these pointers are decremented. The problem with decrementing reference counts of objects referenced by newly freed objects at the time of deallocation is that this results in an unbounded amount

Reference count collectors, however, suffer from three disadvantages: an inability to reclaim circular structures, memory fragmentation, and poor efficiency. Fig. 1.2 illustrates a scenario with an unreclaimable cycle. The reference counts of objects B and C can never fall below 1 as these two objects point to each other. Consequently, B and C are never reclaimed. Such objects are therefore never reclaimed even if there is no path to these objects from the root set.

In reference counting collectors as more and more objects become garbage, free-objects are interspersed with allocated objects. This leads to fragmentation problems. Fig. 1.2 also illustrates fragmentation: note that objects B and C separate the two free objects so that they cannot be coalesced into one large segment of free memory.

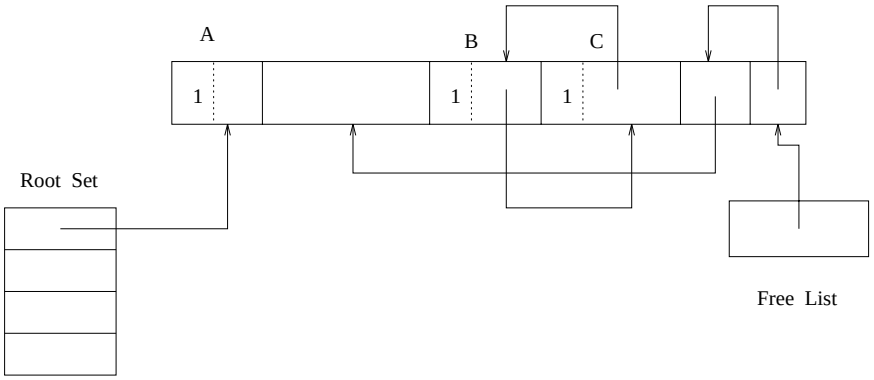


Fig. 1.2: Cyclic Data Structures vs. Reference Counting

The cost of reference counting is normally proportional to the amount of work done by the application program. Whenever a pointer is created or destroyed, its referent's count must be incremented or decremented respectively. An assignment to a pointer field causes two reference counts to be adjusted. If an object's reference count becomes zero, additional work is performed. In general, it is very hard to make reference counting efficient as efficiency is limited by the amount of work done by the application program.

Mark-and-Sweep Collection

These collectors are named after the two phases that implement the garbage collection algorithm. Garbage collection is divided into the following two parts:

1. The *mark* phase traverses all live objects starting from the root set. Two types of traversal are possible: breadth-first or depth-first. Objects reached by this traversal are live and need to be preserved. They are marked as being live either by modifying bits within the corresponding objects or by setting bits in a bit map.
2. The *sweep* phase, invoked after the mark phase, scans through the entire heap in search of unmarked objects. Each unmarked object is placed on to an appropriate free list. Implementations may differ in the number of free lists maintained. Typically, several free lists are maintained, with each list representing free objects of a different size.

As in the case of reference counting, allocation normally involves searching through a free list of objects. The mark-and-sweep technique is illustrated in Fig. 1.3. Fig. 1.3.A shows the normal layout of a heap allocated object and in particular the presence of a *mark bit*. Fig. 1.3.B shows a scenario after the mark phase in which the free list is empty. All the live objects have their mark-bit set to 1. Fig. 1.3.C shows the heap after the sweep phase. As can be seen, the objects with mark bits set to zero after the mark phase, are linked onto the free list. The sweep phase resets the mark bits of live objects to 0.

---

of computation at the time that the memory is deallocated.

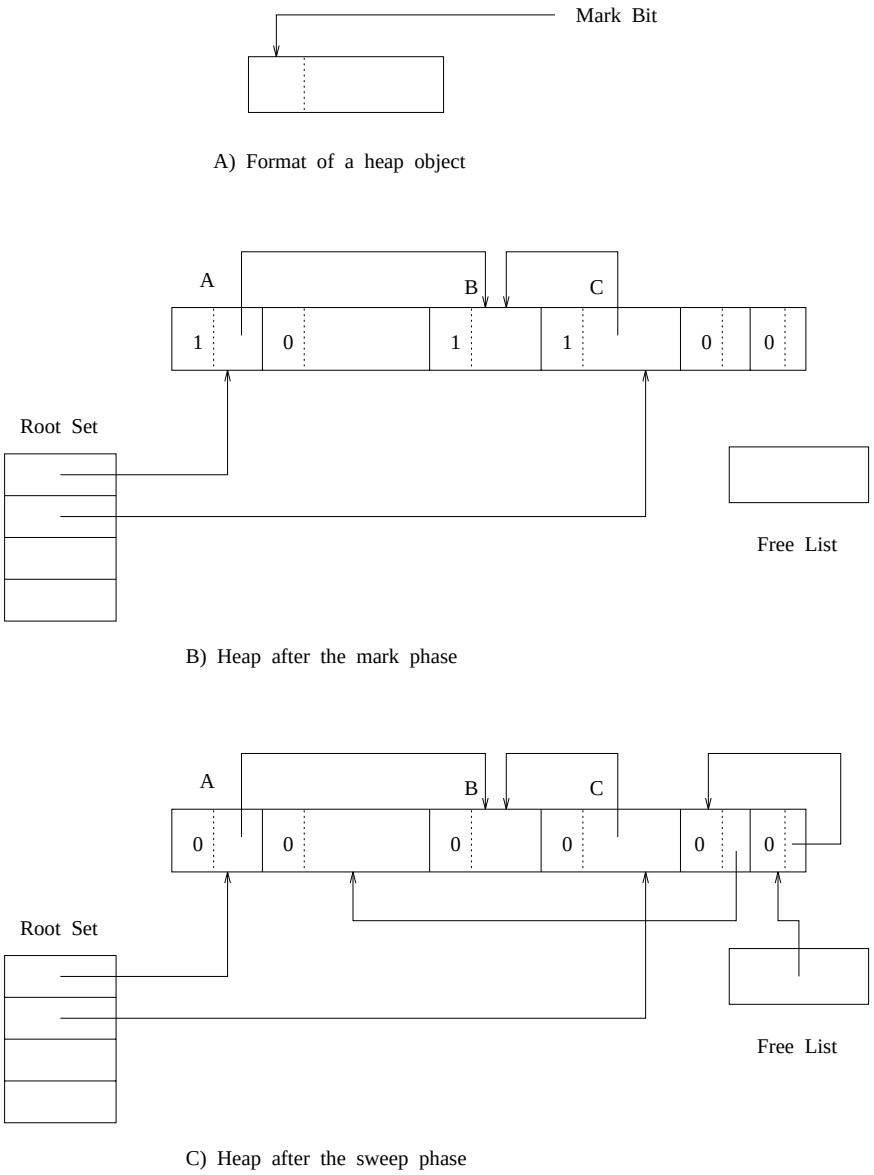


Fig. 1.3: Mark-and-Sweep Collectors

Mark-and-sweep collectors can reclaim circular structures. However, they suffer from the following disadvantages.

1. This technique leads to memory fragmentation. Live objects are not compacted together. As a result, garbage objects, whose space can be reused, are interspersed with live objects. Although free memory is available, in some cases it may not be possible to satisfy an allocation request if the available memory is scattered. This problem can be minimized by coalescing neighboring free objects, but this does not solve the problem of two free objects being separated from one another by an interleaved live object.
2. The cost of garbage collection in case of mark-and-sweep collectors is proportional to the size of the heap. The entire heap has to be swept to reclaim garbage. This may pose performance problems in systems with large heaps.

Mark-and-Compact Collection

These collectors try to remedy the fragmentation problems of mark-and-sweep collectors. The marking phase for garbage detection is similar to the one in mark-and-sweep collectors. Reclamation involves moving the live objects

to one end of the heap until they are contiguous. The remaining memory space in the heap is free and is used to satisfy future allocation requests. This phase is referred to as the *compaction* phase, hence the name *mark-and-compact* collection.

The contiguous free area solves the fragmentation problem. Allocating objects of different sizes can be as simple as updating a pointer. This collection technique, however, suffers from a serious disadvantage. After the marking phase, several passes over the heap are required to compute new locations for live objects, to update pointers to refer to objects' new locations, and to actually move the objects. This garbage collection technique is illustrated in Fig. 1.4. The New pointer points to the start of the free pool.

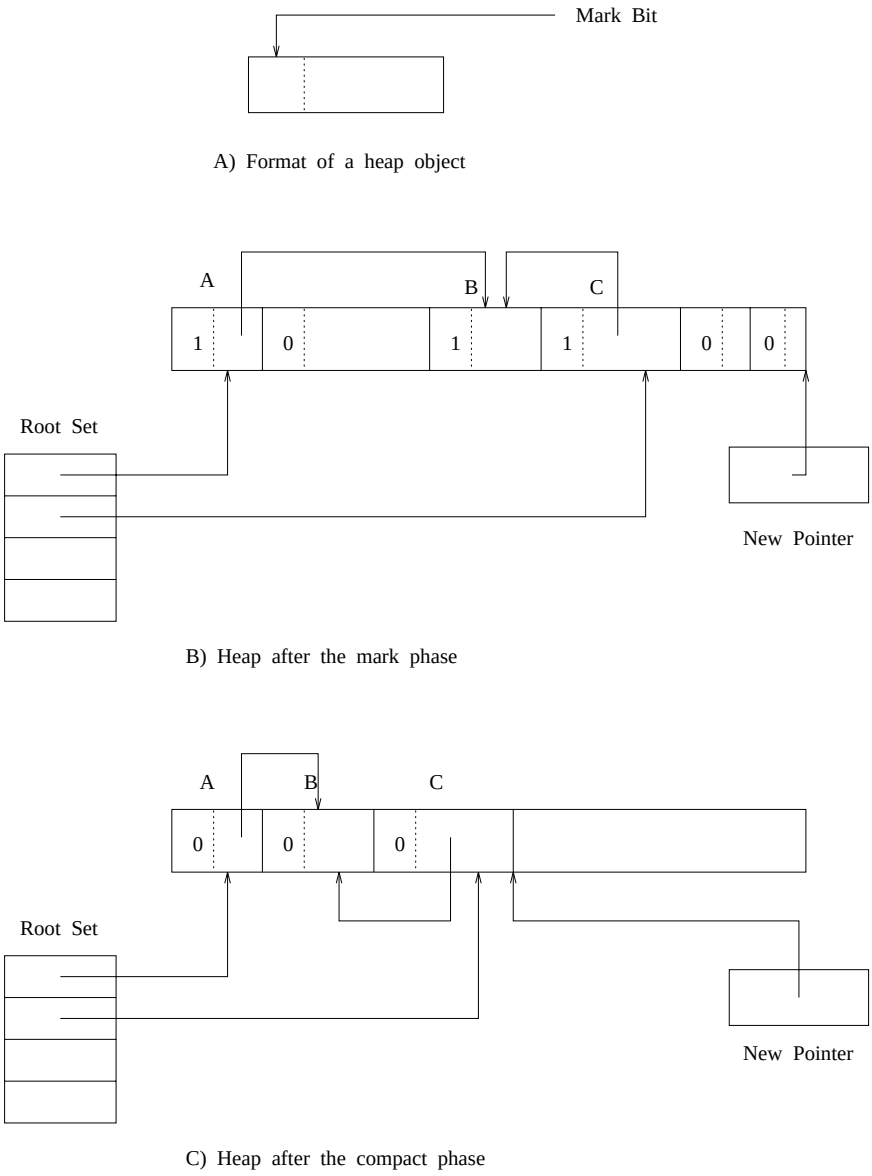
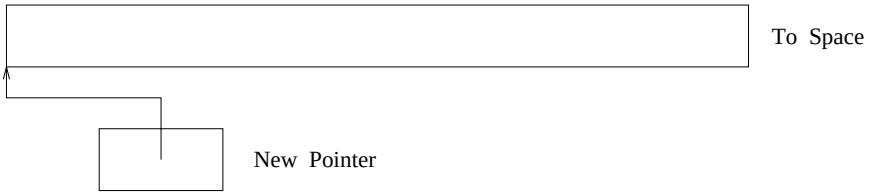


Fig. 1.4: Mark-And-Compact Collectors

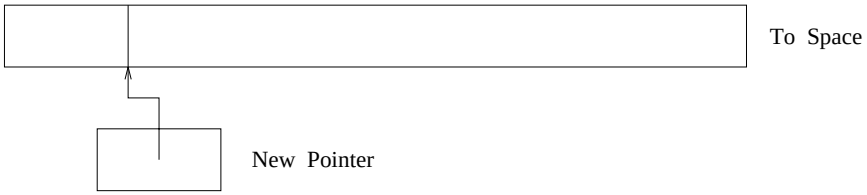
Stop-and-Copy Collection using Semispaces

In this scheme, the space devoted to the heap is subdivided into two contiguous *semispaces*. These semispaces are often referred to as *to-space* and *from-space*. During normal program execution, only *to-space* is in use. All dynamic objects are allocated from *to-space*. Initially, *to-space* contains no objects. Within *to-space*, the New pointer initially points to the beginning of *to-space*. Allocation involves simply advancing the New pointer by size

of the requested object. This is much like allocation from a stack and is therefore fast. Allocation is illustrated in Fig. 1.5.



A) To Space at the beginning of allocation



B) To Space after allocation of an object

Fig. 1.5: Allocation in Semispace Copying Collectors

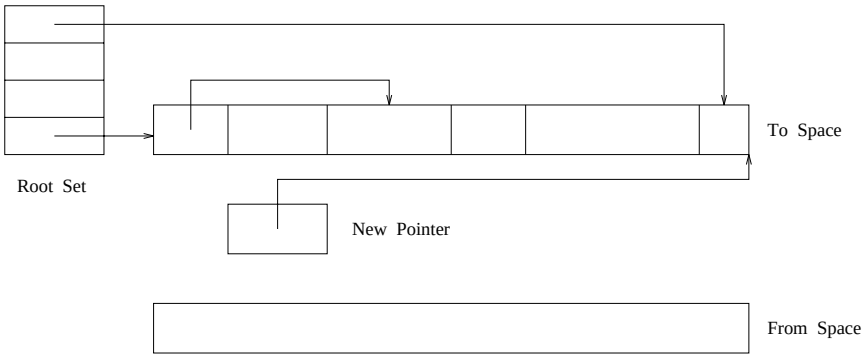
As more and more objects are allocated, the New pointer eventually reaches the end of *to-space*. When this occurs, the application program is stopped and the copying collector is invoked. The functioning of the *to-space* and *from-space* is interchanged by interchanging their names. This is often referred to as a garbage collection *flip*. Garbage collection involves copying all the live objects from *from-space* to *to-space*. A depth-first or breadth-first traversal is used to locate all live objects starting from the root set. When an object is reached during this traversal, space is reserved for this object in *to-space*. A forwarding pointer is set up in the *from-space* copy to point to the reserved space. This is essential to allow the updating of other pointers that refer to the same object. It also prevents multiple scans of an object. All locations known to contain pointers are updated properly to reflect objects' new locations. The *from-space* copy is copied to *to-space* after the object is completely scanned. The space remaining in *to-space* after the copying is available to satisfy future allocation requests. A flip is shown in Fig. 1.6.

A copying garbage collector can be made arbitrarily efficient if sufficient memory is available [15]. The work done at each collection is proportional to the amount of live data at the time of garbage collection. Assuming that approximately the same amount of data is live at any given time during program execution, decreasing the frequency of garbage collections will decrease the total amount of garbage collection effort.

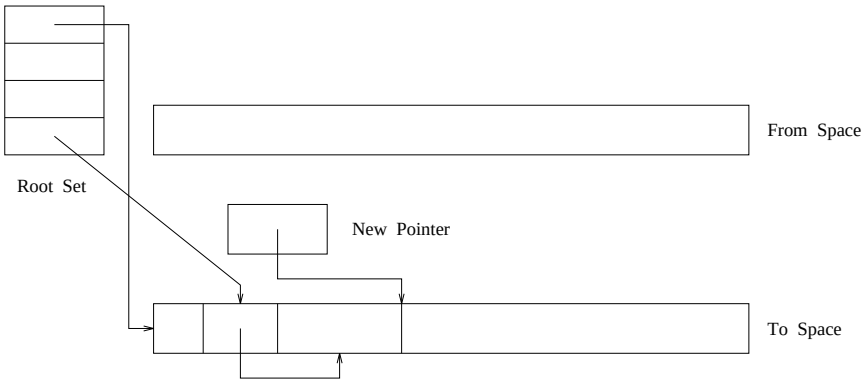
A simple way to decrease the frequency of garbage collections is to increase the amount of memory in the heap. If each semispace is bigger, the program will run longer before filling it. By decreasing the frequency of garbage collections, we also increase the average age of objects at garbage collection time. What this means is that we are increasing the chance for an object to become garbage which also decreases the garbage collection effort.

Generational Garbage Collection

One way to speed up a copying garbage collector is to reduce the amount of storage copied on each garbage collection. *Generational garbage collectors* attempt to do this by treating objects differently depending on their age. They rely on two observations about dynamic storage allocation: new objects are more likely to be freed than old objects, and old objects rarely point to new objects. These garbage collectors ignore long lived objects, the stable set, and concentrate on reclaiming the space occupied by short-lived objects. The heap area is divided into several generations. The younger generation is the smallest in size. It is possible to use different garbage collection algorithms to garbage collect different generations.



A) To and From semispaces at the beginning of garbage collection



B) To and From semispaces at the end of garbage collection

Fig. 1.6: Stop-and-Copy Garbage Collector using Semispaces

In our implementation, the heap is divided into two generations − the *nursery* and the *old generation*. The semispace copying technique is used to garbage collect the nursery and the mark-and-sweep technique is used for garbage collecting the old generation.

Objects are allocated in the nursery. Allocation is similar to the technique shown in Fig. 1.5. When the *to-space* of the nursery is full, the nursery is garbage collected copying its live data into the other semispace (the new *to-space*). An object that survives a threshold number of flips is copied into the old generation. This is often called *promotion* or *tenuring*. Since relatively few objects live long enough to be considered old, the old generation fills up much more slowly than the new generation. Eventually, when the old generation is full it needs to be garbage collected as well. In general, the number of generations may be greater than two, with each successive generation holding older objects and being garbage collected less often.

In order for this scheme to work, it must be possible to garbage collect the new generation without scanning the old generation. Since liveness is a global property, it is possible that some old objects point to new objects. References from the old generation to the new generation are referred to as *cross-generational references*. The nursery maintains a table of such references. This table is called the *remembered set* and has pointers to old objects which refer to objects in the nursery. When we garbage collect the new generation, the objects pointed to by the pointers in the remembered set are also scanned to determine liveness. All the stores to the old generation need to be checked for cross-generational references and in case of one, the object in question should be added to the remembered set. Our implementation is described in more detail in sections 3 and 4.

Within the framework of generational strategy, several tunable configuration parameters exist:

Advancement Policy:

When to consider an object as old?

Heap Organization:
How many generations should the heap be divided into? What algorithm(s) should be adopted for garbage collecting these individual generations?

Cross-Generational References:
What is the best way to keep track of cross-generational references?

Locality:
What effect does promotion have on locality of reference?

2. Compiler Support for Garbage Collection

During the process of garbage collection, the collector relocates live objects. For correct operation of the collector, the collector needs to know where exactly pointers reside in the heap, the static area, the activation stack, and the machine registers. An incorrect interpretation of a value as a pointer might cause memory which is actually unreachable to be retained across collections, and worse might end up changing such a value leading to unpredictable results.

2.1. Signature of a Type

With each basic type or user-defined class, the modified compiler associates a data structure called the *signature* for that type. Signatures are used to locate pointers in heap-allocated objects. All heap-allocated objects have a type and are associated with the signature for the corresponding type. To be more specific, the header portion of a heap-allocated object stores a pointer to its signature.

The first word of the signature is an integer that represents the size of an object of the corresponding type, measured in words. Following this, the signature stores the number of words that must be scanned by the garbage collector. If the object contains no pointers, this value is zero. Following this, there is one 32-bit tag word for each group of 32 words that must be scanned by the collector. These tag bits are used to distinguish between data words in the object that contain pointers and those that do not. The bit is on if the corresponding data word holds a pointer, and off otherwise.

For every type declared in a program, the compiler generates a signature representation. To illustrate, consider the following C++ class declaration:

```
class Tree {
  Tree *lptr;
  Tree *rptr;
  int val;
};
```

As can be seen from the declaration for the class Tree, an object of type Tree will contain pointers in the first two words and a non-pointer in the third. The signature generated by the compiler for this type would thus be:

```
int Tree_signature[] = {
  3,        // Total size is three words
  2,        // We need to scan two words
  0x3           // First two words store pointers (binary: 011)
};
```

More work needs to be done in the case of unions within which particular words may, at different times, serve as both pointers and non-pointers. When generating the signature for such a union, the compiler generates a default signature assuming that such words do not store any pointers. To illustrate, consider the following C++ union:

```
union foo {
  int i;
  int *ip;
};
```

The default signature generated would be:

```
int foo_signature[] = {
  1,        // Total size is one word
  1,        // Scan 1 word
  0x0           // Assume no pointers initially
};
```

Assignments to non-union fields are the same as for traditional code. However, assignment to any union field that can potentially represent either pointer or non-pointer data, has considerably more overhead than in the traditional implementation. To illustrate, consider the following C++ code segment:

```
foo  *fp;
int  i;

fp = new foo;
fp->ip = &i;
```

The assignment to fp->ip, in addition to performing the required assignment, must also change the signature associated with the object referenced by fp to:

```
int fp_signature[] = {
  1,        // Total size is one word
  1,        // Scan 1 word.
  0x1           // Object pointed to by fp now stores a pointer
};
```

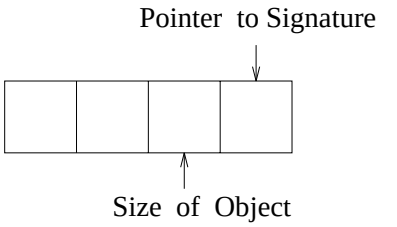
It is to be noted that when the signature of an object changes dynamically, as in the previous example, the signature is stored as part of the corresponding object. The signature pointer in the header, in such a case, points to the location within the object that stores the signature.

To illustrate how signature is stored as part of heap allocated objects, consider the following code segment.

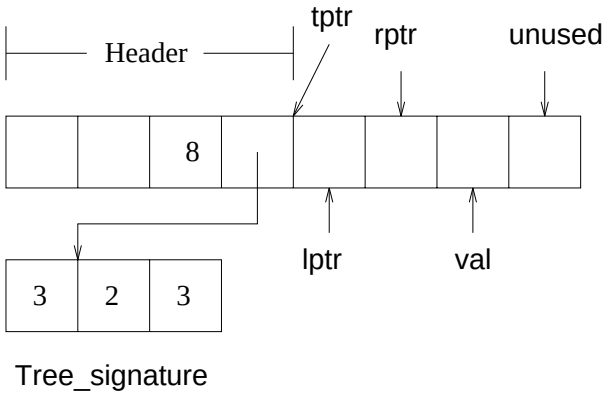
```
Tree *tptr;
foo *fptr;

tptr = new Tree;            // Object with static signature
fptr = new foo;             // Object with dynamic signature
```

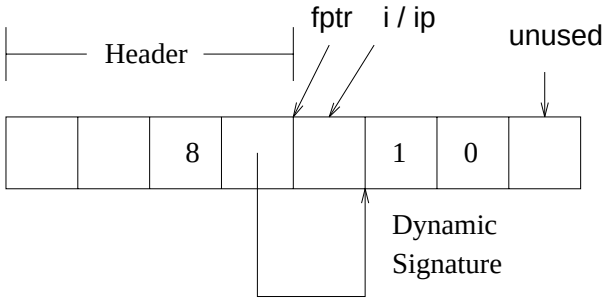
The heap allocated objects, for these allocations are shown in Fig. 2.1.1. Fig. 2.1.1.A shows the layout of object headers. An object header contains four words. The third word of the header stores the size of the heap allocated object and the last word a pointer to the object's signature. In our implementation, all heap allocated objects are aligned on four-word boundaries and have a four-word header. The size field in the header of the object allocated to tptr, as a result, stores a value of 8 (as shown in Fig. 2.1.1.B). An object of type Tree has a static signature. The pointer-to-signature field in the header, as a result, points to the common signature of all objects of type Tree. The object allocated to fptr, however, has a dynamic signature. It can be seen from Fig. 2.1.1.C that the signature is stored as part of the allocated object. The pointer-to-signature field, in this case, points to the location within the object starting where we store the signature.



A) FORMAT OF OBJECT HEADER



B) HEAP ALLOCATED OBJECT WITH STATIC SIGNATURE



C) HEAP ALLOCATED OBJECT WITH DYNAMIC SIGNATURE

Fig. 2.1.1 Heap Objects

2.2. Changes to the Register Allocator

For the purpose of garbage collection, the general purpose registers were partitioned into two classes.

1. Registers which store only pointers − *pointer registers*.
2. Registers which store only non-pointers − *non-pointer registers*.

To illustrate, consider the SPARC architecture with 32 general purpose registers r0 through r31. A typical partition might look like:

Pointer registers	r0 to r3, r8 to r11, r16 to r19, and r24 to r27.

Non-pointer registers	r4 to r7, r12 to r15, r20 to r23, and r28 to r31.

The register allocator honors this partitioning of the general purpose registers. It makes sure that only pointers reside in pointer registers, and only non-pointers reside in non-pointer registers. The garbage collector scans only the pointer registers to determine the liveness of heap-allocated objects.

2.3. Activation Frame Layout

The typical layout of an activation frame is illustrated in Fig. 2.3.1.

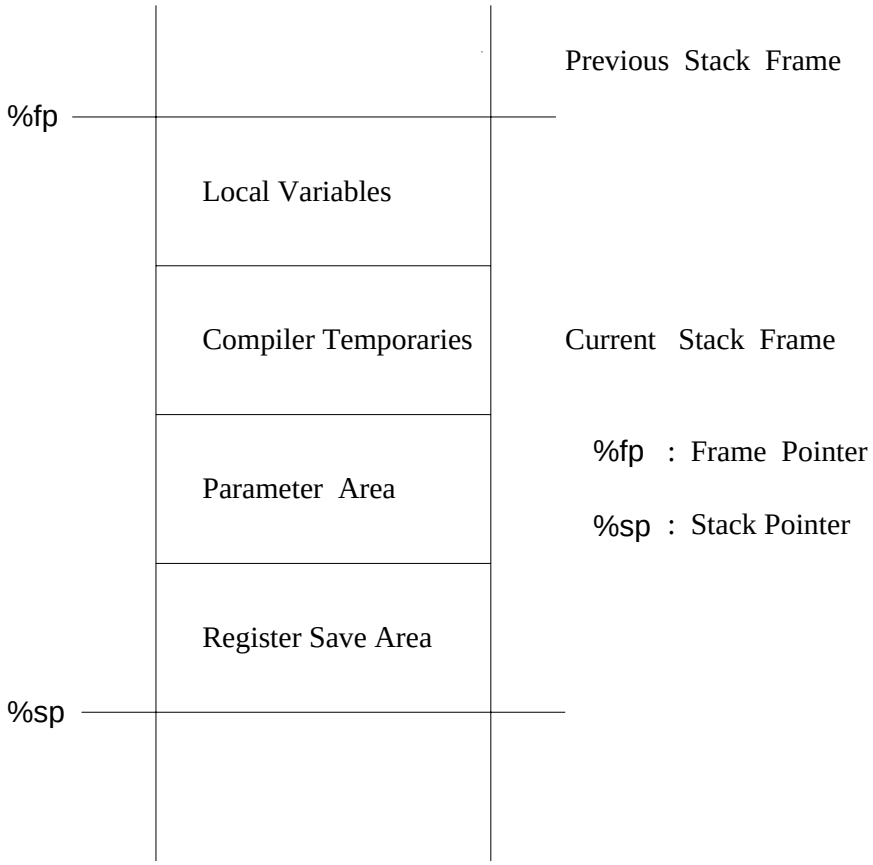


Fig. 2.3.1 Typical Activation Frame Layout

To illustrate how the activation stack changes on a function call, consider the following code segment.

```
foo() {
  baz();
}

baz() {
  int i1, i2, i3;
  foo *fp1, *fp2;
  int i4, i5;

  function body ...
}
```

The layout of the activation stack after the call to baz() is shown in Fig. 2.3.2. If garbage collection is triggered when the application is executing a statement in function baz, the garbage collector needs to know exactly which locations within the activation frame store pointers. The organization of activation frames as shown in Fig. 2.3.2 makes it difficult to determine this information efficiently.

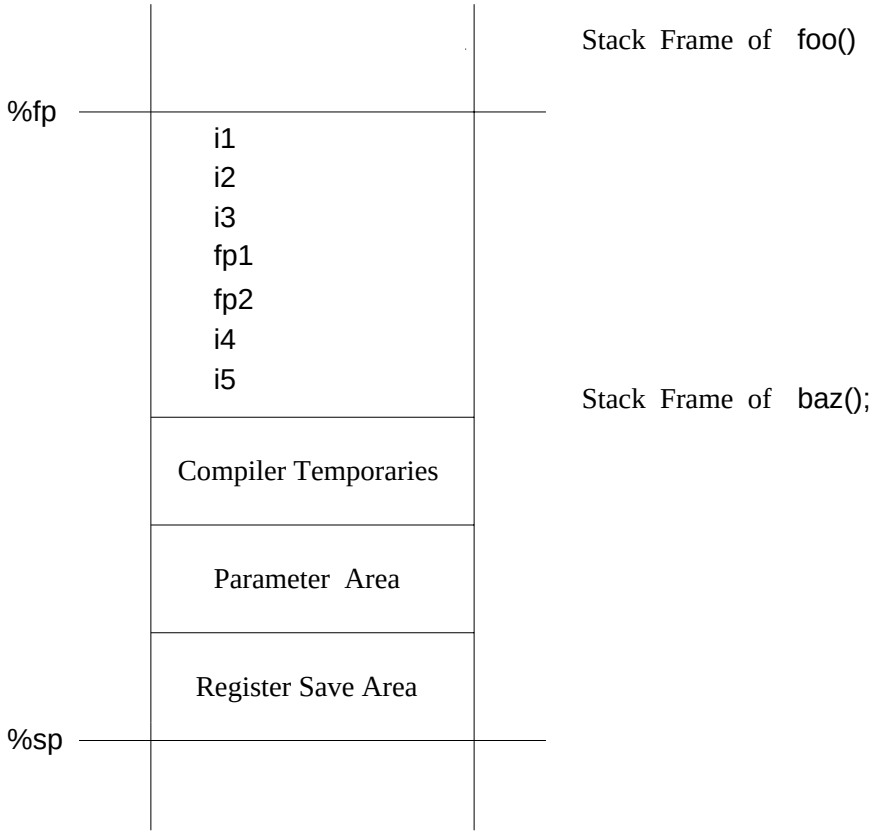


Fig. 2.3.2: Activation Stack After the Call to Function baz()

We propose a new design for stack activation frames which divides the stack into *segments* of four words each, with alternating segments storing pointers and non-pointers. This organization is illustrated in Fig. 2.3.3. The activation stack, for the example given earlier, would be as shown in Fig. 2.3.4 if we use this new stack organization. Note that this scheme allows the garbage collector to quickly and efficiently determine exactly which words within the stack hold pointers.

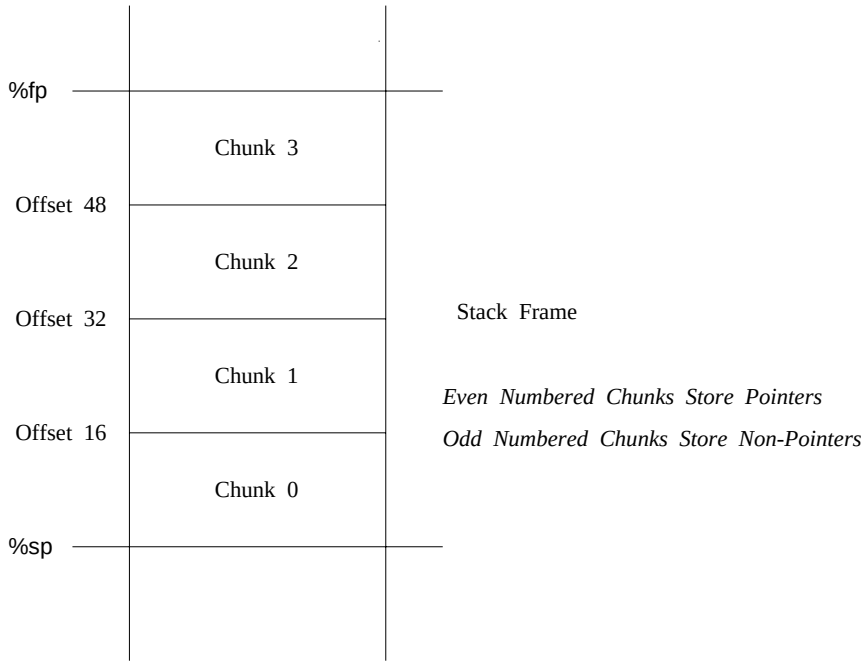


Fig. 2.3.3: New Activation Frame Layout

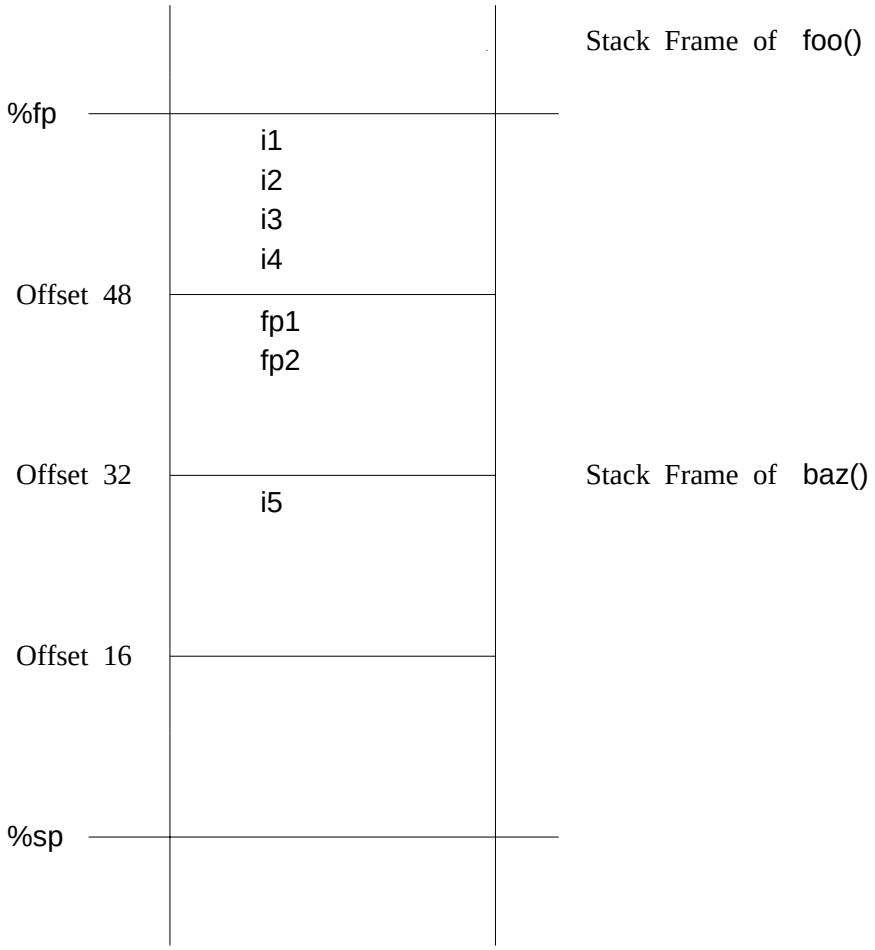


Fig. 2.3.4: Revised Activation Stack After Call to Function baz()

Note also that there is no place in this new stack organization for representation of certain aggregate data types, such as an array of ten integers. Functions whose local variables fall into such a category are associated with an area in the heap called the *aggregate area*. This area stores all of the local variables that do not fit suitably into the stack-allocated activation frame. A signature is associated with the aggregate area to identify the location of pointers in this area. A pointer register is dedicated to point to this area in the heap. To illustrate, consider the following code segment:

```
class str {
  int c;
  int *ip;
  int b;
}

foo1() {
  baz1();
}

baz1() {
  int i1, i2, i3;
  foo *fp1, *fp2;
  int i4, i5;
  int arr[10];
  str s1;

  function body ...
}
```

The stack layout for this example is shown in Fig. 2.3.5.

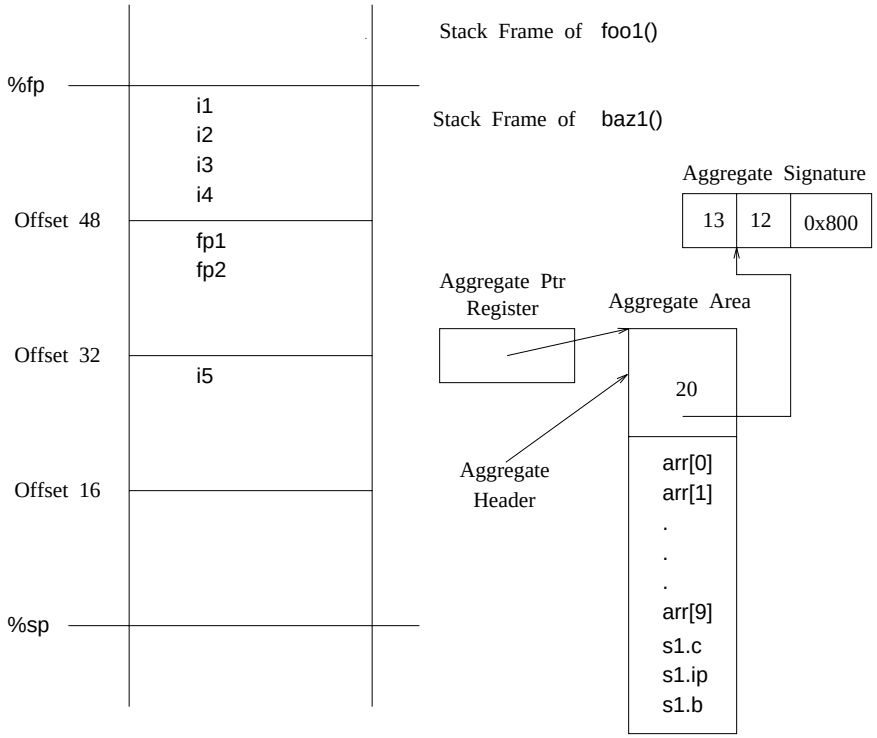


Fig. 2.3.5: Revised Activation Frame for a Function with Aggregates

2.4. The gcdata record

The lifetime of file-scope and static variables in C++ is the duration of program execution. Since there may be any number of such variables, and since they may be of any type, there is no limit on the number of pointers into the heap that may be contained within these regions. To assist the garbage collector, all file-scope and static variables are collected into a large record, called the *gcdata* record. A signature is associated with this record. The address of this record can be either stored in a well known location or in a dedicated pointer register. To illustrate, consider the following global declarations in program foo:

/* Type declarations */	/* Globals */

class s1 {
  int c;
  int b;
  int *ip;
};
 	
class s2 {
  s1 *s1p;
  s2 *next;
  int val;
};
	s1 g1;
	s2 g2;
	int g3;
	int g4;
	int *g5;
	int g6[10];

The gcdata record, for program foo, will be as shown in Fig. 2.4.1.

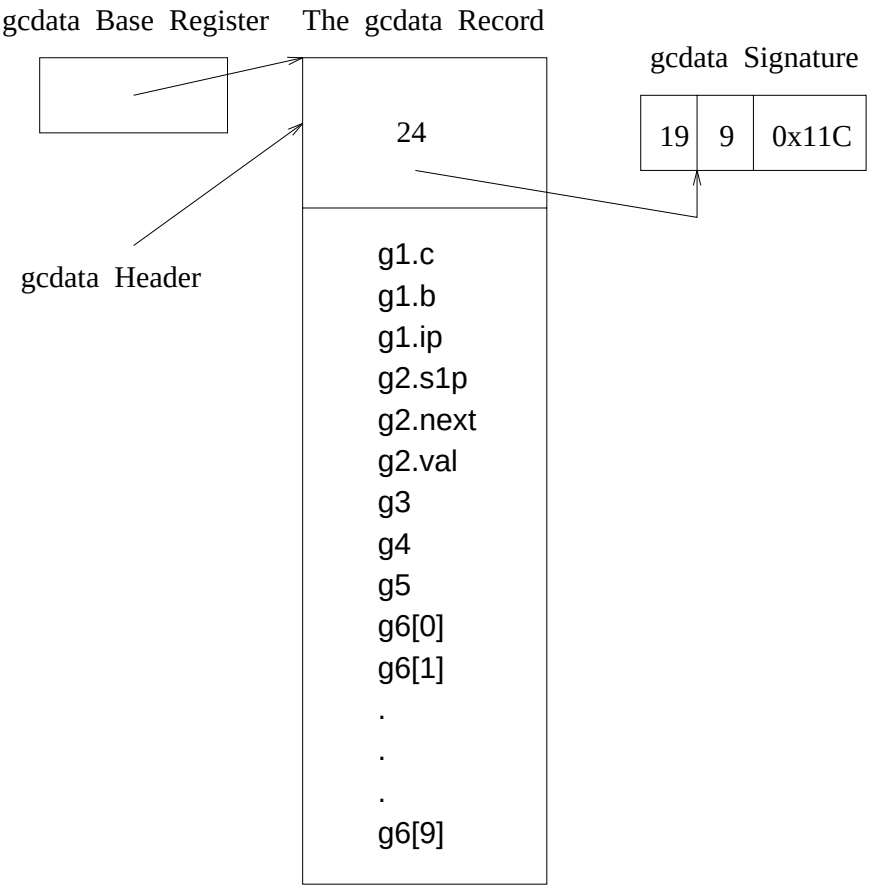


Fig. 2.4.1: The gcdata record for program foo

3. Allocation of Heap Objects

As mentioned before, the garbage collector handles all heap allocations. The heap area is divided into generations. This layout is described in section 3.1. The header portion of heap-allocated objects stores information that helps in locating pointers in the heap-allocated object and in determining the object's size. Given a pointer to a heap-allocated object, it is possible for the collector to locate the header of the corresponding object. A heap allocation map is maintained to achieve this. We describe the heap allocation map and object headers in section 3.2. The allocation routines are described in section 3.3.

3.1. Layout of the Heap

For the purpose of garbage collection, the heap is divided into two generations − the *old generation* and the *new generation*, also known as the *nursery*. The nursery is further divided into two semispaces − the *to-space* and the *from-space*. This layout is shown in Fig. 3.1.1. The old generation is organized as a collection of linked lists of free objects. Fig. 3.1.1 illustrates a single free list that links all of the objects in the the old generation. The mark-and-sweep technique is used to garbage collect the old generation. New objects are allocated in the nursery and the semispace copying technique is used to garbage collect the new generation.

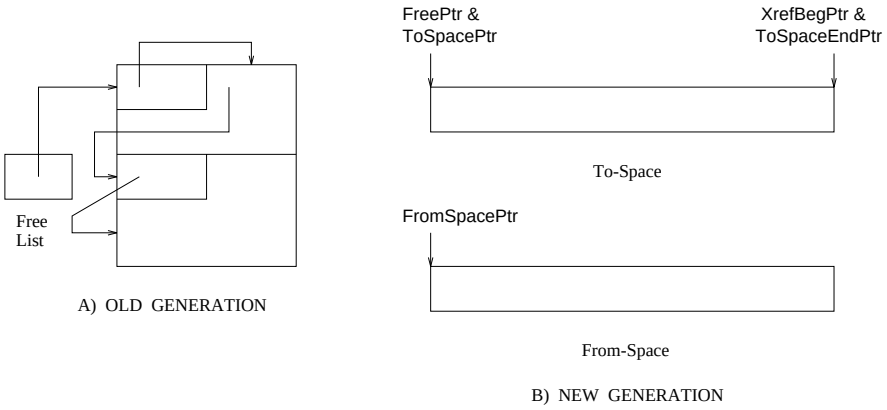


Fig. 3.1.1: Organization of the Heap

In the discussion that follows, *to-space* and *from-space* refer to the appropriate segments of memory within the nursery.

The following pointers to the nursery are of interest.

ToSpacePtr:
: Points to the beginning of *to-space*.

FromSpacePtr:
: Points to the beginning of *from-space*.

ToSpaceEndPtr:
: Points to the end of *to-space*.

FreePtr:
: Points to the beginning of the free pool found within *to-space*.

XrefBegPtr:
: Points to the location in *to-space* starting where we store the cross-generational references. The locations between XrefBegPtr and ToSpaceEndPtr store the addresses of objects in the old generation which may hold cross-generational references. When an application is started, XrefBegPtr points to the end of *to-space*.

3.2. Heap Allocation Map and Object Headers

All heap-allocated objects are aligned on 16-byte boundaries. The heap allocation map is used to report the address of the first data word of each heap-allocated object. To do this, the heap allocation map dedicates one bit to each 16-byte group. The bit is set if a heap-allocated object begins at the corresponding 16-byte group.

The garbage collector provides the following interface to the heap allocation map.

void CreateObject(int *ip):
: This routine sets a bit in the heap allocation map to indicate that an object begins at address ip.

void DeleteObject(int *ip):
: This routine clears a bit in the heap allocation map to indicate that an object no longer begins at address ip.

void *findObjectHeader(int *derivedptr):
: Given a pointer to a location within an object - derivedptr, this routine is used to report the address of the *first*

data word of the object pointed to by derivedptr. This routine searches backwards through the heap allocation map, starting at the bit corresponding to the location pointed to by derivedptr, looking for a bit whose value is set. Once it finds a bit that is set, it returns the corresponding address.

void ClearWords(int *ip, int numwords):
This routine is used to clear the heap allocation map. It clears 'numwords' words starting at address ip. This routine is invoked at the start of an application and after every *flip* to clear the allocation maps.

Preceding the first data word, heap-allocated objects, have a four word header. The header typically stores the following information.

1. Size of the corresponding heap-allocated object.
2. A pointer to the corresponding object's signature.

These two fields are initialized during the allocation process. In addition to these fields, the header is also used to store the following information during the garbage collection process.

Forwarding-Pointer:
The garbage collector relocates live objects during the collection process. The forwarding-pointer in the header points to the location to which the corresponding object is going to be moved after collection.

Age: This field is used to keep track of the number of garbage collections that an object survived.

Marklist:
This field is used to maintain a linked list of objects that are to be scanned by the garbage collector.

3.3. Heap Allocation

The application makes a request for dynamic memory using the new statement. On encountering a new statement, the compiler generates code to make a call to the allocation routines provided by the collector.

3.3.1. Allocation of a Single Object

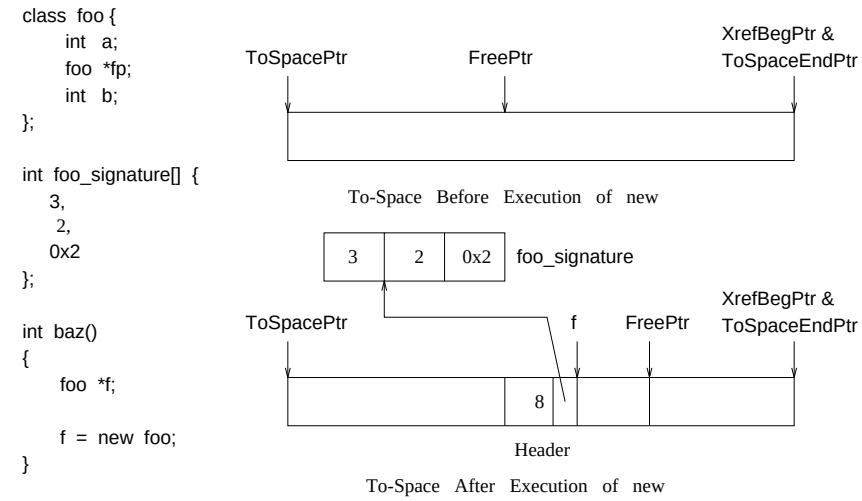
A call to the allocation routine, in this case, looks like:

```
gcalloc(size, ptrtosig, dynamic)
```

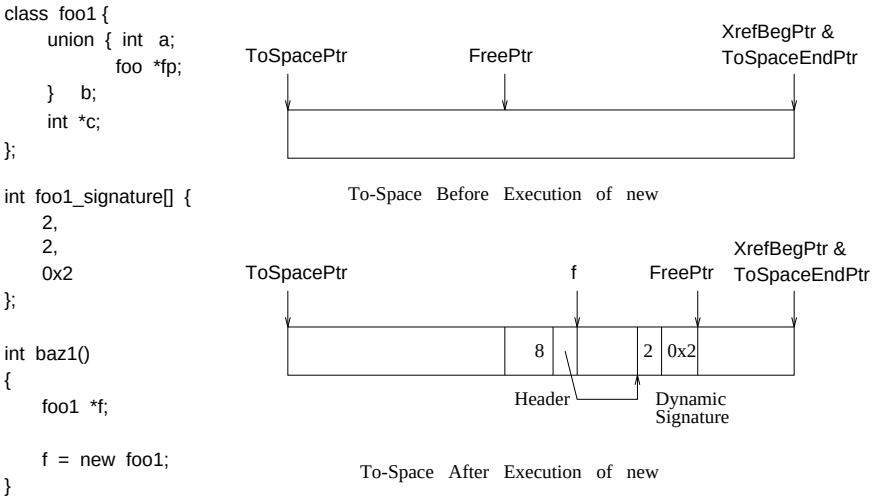
The first argument to the allocation routine gives the size of memory required in bytes. The second argument gives the signature to be associated with the new object being allocated. The dynamic flag is used to distinguish between static and dynamic signatures. It is non-zero if the signature changes dynamically (if the object includes a union that contains both pointer and non-pointer variants). The steps performed by the allocation algorithm are outlined below:

1. Compute the number of words to be allocated by adding up the number of words required for the header; the signature, if it is needed; and the requested object.
2. Round up the calculated number of words to the next multiple of four. This is required as all objects are aligned on 16-byte (4-word) boundaries.
3. Check if enough memory is available in the *to-space*. Enough memory is available if XrefBegPtr − FreePtr is at least as big as the required number of words.
4. If enough memory is not available, perform garbage collection. If garbage collection fails to reclaim sufficient memory to satisfy the request, return a failure condition to the caller.
5. Set up the size and pointer to signature in the header.
6. If the signature is dynamic, store a copy of the signature as part of allocated object. Omit the total size field from the copied signature, since this is redundant with information already contained within the object's header.
7. Initialize all pointers within the allocated object to NULL.
8. Increment freeptr by the number of words allocated.
9. Return the address at which the newly allocated object begins.

Allocation is illustrated in Fig. 3.3.1.1.



A) ALLOCATION OF AN OBJECT WITH STATIC SIGNATURE



B) ALLOCATION OF AN OBJECT WITH DYNAMIC SIGNATURE

Fig. 3.3.1.1: Allocation of heap objects

3.3.2. Allocation of Arrays

The prototype for the array allocation routine is shown below:

gcallocarr(int size, int *ptrtosig, int numelems)

In this prototype, size represents the number of bytes in each array element. If padding is necessary between array elements, this should be represented by the value of the size parameter. The ptrtosig argument points to the type signature for each array element. The numelems argument specifies the number of elements in the array to be allocated.

The allocation algorithm is similar to the one given in section 3.3.1 except that step 6 is replaced with the following:

6': The signature of the allocated array is considered to be dynamic. Space is reserved for a dynamic signature within the allocated object following the memory set aside to represent the array contents. The object's signature is computed by replicating the bitmap found within *ptrtosig the appropriate number of times.

The allocation of an array object is illustrated in Fig. 3.3.2.1.

```
class foo {
      int  a;
      foo  *fp;
      int  b;
};

int foo_signature[] {
      3,
      2,
      0x2
};
```

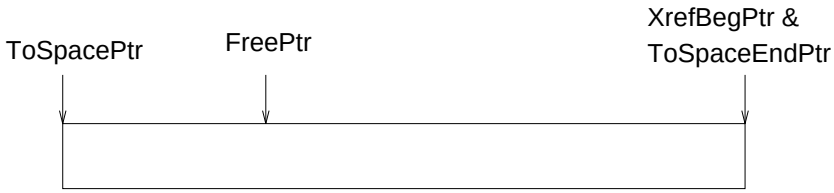
```
int baz()
{
     foo  *f;

     f = new foo[5];
}
```

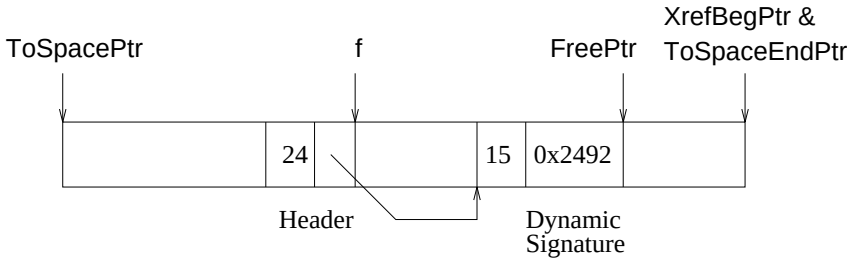
A) CODE SEGMENT WITH AN ARRAY ALLOCATION

```
int foo1_signature[] {
     15,                  // Total size of this array object
     15,                  // Scan 15 words
      0x2492              // Binary  010 Concatenated 5 times
};
```

B) EXTENDED SIGNATURE FOR ARRAY OBJECT, CONSTRUCTED DYNAMICALLY



To-Space Before Execution of new



To-Space After Execution of new

C) ALLOCATION OF AN ARRAY OBJECT

Fig. 3.3.2.1: Allocation of a Dynamic Array

4. The Garbage Collection Algorithm

In this section, *root set* means the set of pointers in the activation stack, the machine registers, and the global area. Note that the changes to the GNU C++ compiler make it straightforward to locate all of the pointers in the root set.

Generational garbage collectors try to concentrate most of their effort in garbage collecting the younger generation. Generational collectors avoid scanning the old objects. To enable the garbage collector to determine all the live objects in the nursery, without actually scanning through the entire old generation, the garbage collector maintains the addresses of objects in the old generation that may hold pointers to the nursery. One method for doing this is described in section 4.1. The garbage collection algorithm for the nursery is given in section 4.2. We give the collection algorithm for the old generation in section 4.3.

4.1. Maintaining Cross-Generational References

The compiler keeps track of all writes into the old generation. When a value is written to a location in the old generation, known to be of type pointer, the compiler checks if the value being written is a pointer to the nursery. If the value being written is a pointer to the nursery, the compiler decrements the value of XrefBegPtr and stores the address of the modified object in the location pointed to by XrefBegPtr. Fig. 4.1.1 illustrates this process. The set of objects pointed to by locations in the range between XrefBegPtr and ToSpaceEndPtr constitute the *remembered set*.

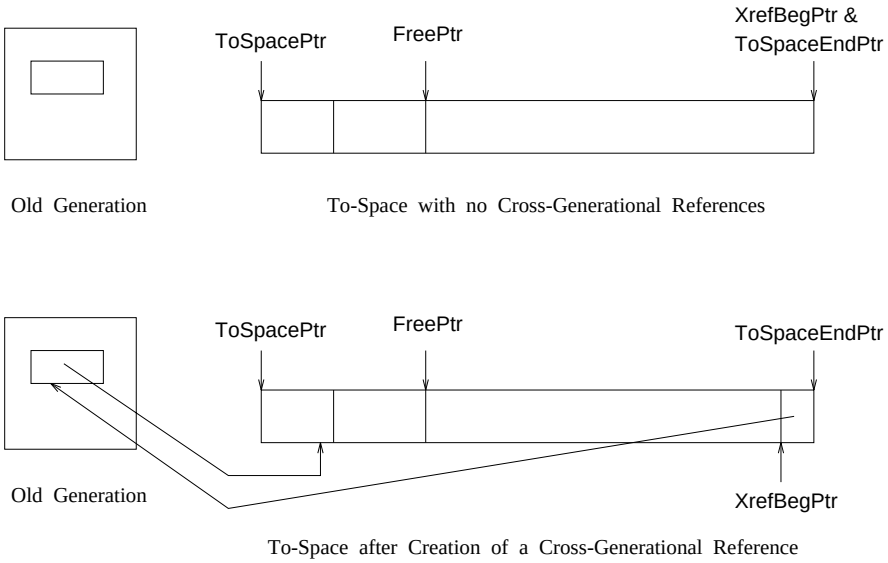


Fig. 4.1.1: Cross-Generational References

4.2. Garbage Collection Algorithm for the Nursery

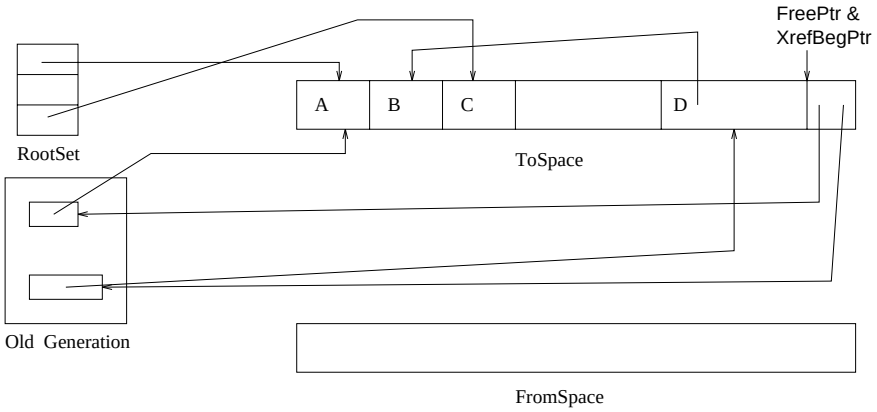
In our implementation, garbage collection is triggered when a request for memory cannot be satisfied. The garbage collection algorithm for the nursery proceeds as follows:

1. Copy the pointers found between XrefBegPtr and ToSpaceEndPtr to the end of the current *from-space*. Set XrefBegPtr to point to the start of the remembered set in *from-space*[3].
2. Swap the *to-* and *from-spaces*. The current *from-space* becomes the new *to-space* and vice-versa. Update relevant pointers (ToSpacePtr, FromSpacePtr, ToSpaceEndPtr) accordingly. Set FreePtr to point to the beginning of the new *to-space*.
3. Make a linked list (mark list) of nursery objects that are directly reachable from the root or remembered sets[4].
4. When adding an object to mark list, reserve space for the object in either the old generation (if the object is old enough) or in the *to-space*. Set up forwarding-pointer in the header to point to this reserved space. Modify the pointers in the root and remembered sets to point to the new locations of objects.

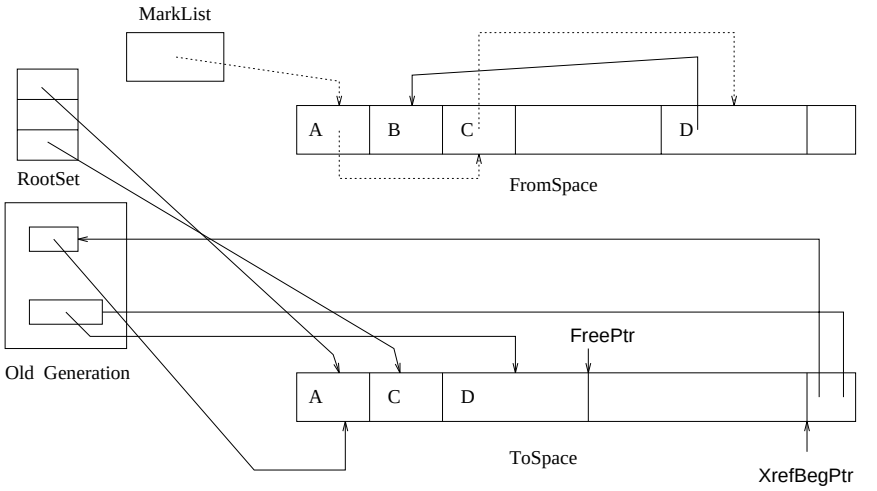
[3] Note that we copy the entire remembered set, even though some of the addresses in the remembered set may no longer hold cross-generational pointers. A future improvement to the implementation will copy only those pointers that still describe memory cells containing pointers into the nursery.

[4] Unlike techniques described previously [11], this copying garbage collection implementation is not incremental. The choice to build a mark list instead of incremental copying was an accident of development. There is no reason why this system could not use the more traditional incremental copying technique.

5. If unable to promote an object to the old generation, set a flag to indicate that it is time to garbage collect the old generation. Delay the promotion.
6. For every object on the mark list
   a. Scan through the object word by word until the last word known to contain a pointer. While scanning, if a location stores a pointer to a new object, check if the forwarding-pointer is set in the object pointed to by this pointer.
   b. If the the forwarding pointer is not set link this unmarked new object onto the mark list. In the process, the forwarding pointer will be set for this object.
   c. Modify the pointer in the object being scanned to point to the new location.
   d. Once an object is scanned, copy the header portion to the space reserved for the object. Clear the forwarding-pointer. As we scan through the object, the corresponding locations in the reserved space are set appropriately. Copy over the unscanned portion of the object.
7. If time to garbage collect the old generation, garbage collect. Reset the flag.



A) HYPOTHETICAL SITUATTION - ToSpace IS FULL

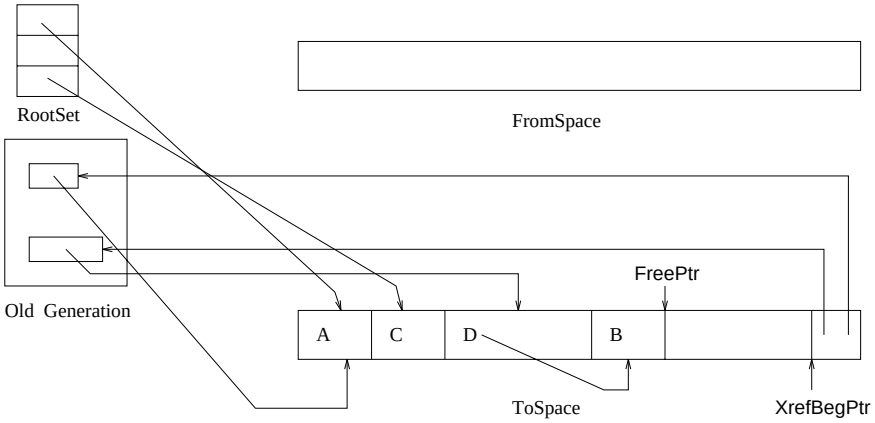


B) AFTER SCANNING POINTERS IN ROOTSET AND XREFOBJECTS

Fig. 4.2.1: Garbage Collection of the Nursery

The garbage collection of nursery is illustrated in Figs. 4.2.1 and 4.2.2. Fig. 4.2.1.A shows a hypothetical situation in which the *to-space* is full. We have two pointers from the root set to objects in nursery and two objects which contain cross-generational references. Fig. 4.2.1.B shows the situation after adding the objects directly reachable from the

root and remembered sets to the mark list, and after updating all of the pointer values in these sets. Fig. 4.2.2 shows the situation after complete garbage collection.



AFTER COMPLETE COLLECTION OF THE NURSERY

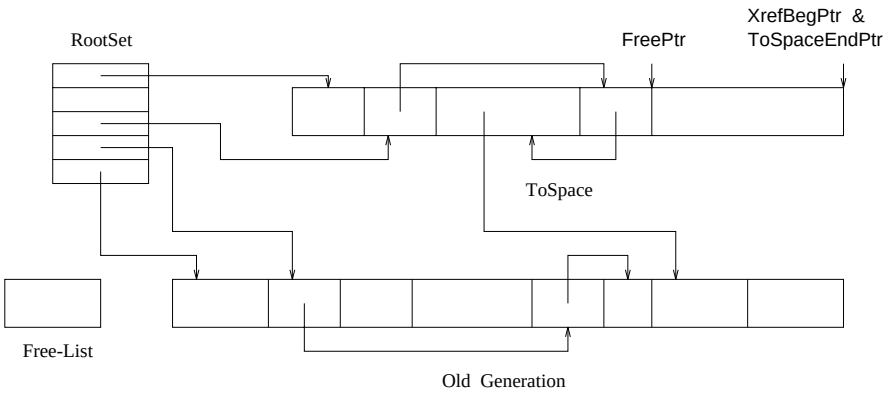
Fig. 4.2.2: Garbage Collection of the Nursery

4.3. Garbage Collection Algorithm for the Old Generation

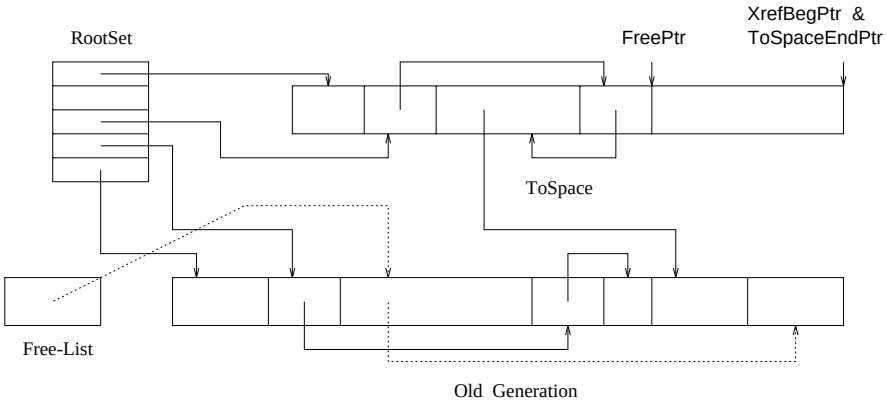
Garbage collection of the old generation is triggered when a request for promotion of an object to the old generation fails. As described in section 4.2, the garbage collection of old generation is delayed until the completion of garbage collection of nursery. Once the garbage collection of nursery is complete, all the objects which reside between ToSpacePtr and FreePtr are live. Based on this fact, the garbage collection algorithm for the old generation proceeds as follows:

1. Initialize the remembered set for the current *to-space* to empty.
2 Make a linked list (mark list) of old objects (objects in the old generation) that are directly reachable from the root set and objects in the *to-space*. The objects on the mark list are said to be marked.
3. For every object on the mark list:
   a. Scan each pointer word within the object.
   b. While scanning, if a location is a pointer to an old object which is unmarked, add the unmarked object to mark list.
   c. While scanning, if a location is a pointer to a nursery object, add the address of this word to the cross-generational log.
3. Sweep through the old generation looking for unmarked objects.
4. During the process of sweeping, unmarked objects are added to the free-lists that are maintained. Neighboring objects that are unmarked are coalesced to reduce fragmentation.

Garbage collection of old generation is illustrated in Fig. 4.3.1. Figure 4.3.1.A gives a scenario where the free list is empty. Note that garbage collection of the old generation can be triggered even if there are objects on the free list. This might be caused if none of the available objects is big enough to satisfy a request for promotion. In Fig. 4.3.1.B we have shown the situation after garbage collection of the old generation. The unmarked objects have been added to the free list and neighboring unmarked objects (free objects) have been coalesced.



A) A SCENARIO AFTER COLLECTION OF THE NURSERY



B) AFTER COLLECTION OF THE OLD GENERATION

Fig. 4.3.1: Garbage Collection of the Old Generation

5. Results and Scope for Future Work

In this section we present the preliminary results that we have obtained. Because various changes to the GNU C++ compiler have not yet been totally integrated, it is not yet possible to measure the performance of complete C++ programs. Once the changes to the compiler have been integrated, we plan to test our garbage collector with real applications. We have tested our generational garbage collector using test cases which did not require us to scan through the activation stack and the machine registers. Section 5.1 presents the timings we obtained on garbage collections of the nursery. In section 5.2, we discuss our results obtained on complete collections. Section 5.3 concludes the report with our suggestions for future work.

5.1. Garbage Collections of Nursery

For the test cases described in this section, the old generation size was fixed at 256 kbytes. The size of the nursery was varied and we collected information such as the number of times the nursery was garbage collected and the total execution time using the gprof utility. For comparison purposes, we also measured the total execution times for these test cases using malloc and free in place of the allocation routines provided by the collector.

The first experimental workload repeatedly builds a large linked list and then discards the first half of the objects in the linked list. Since consecutive free operations release objects allocated by consecutive malloc invocations, most of the free calls are very efficient. This is because each discarded object can be coalesced with the previously discarded object. The application requests a total of 528 kbytes and frees a total of 512 kbytes. The test results for this example are shown in Table 5.1.1. The malloc/free version of this program took 2.50 seconds to execute and used 43 kbytes of

memory. The same application took 2.17 seconds to execute when the calls to free were commented out. In this case, however, the application ended up using 1067 kbytes of memory. The same application took 2.99 seconds to execute when only a portion of the allocated memory was freed using free. The application in this case used 556 kbytes of memory.

Total Heap Size (KBytes)	*Size of Nursery (KBytes)*	*Number of Collections of Nursery*	*Total Time (Seconds)*
387	64	64	2.94
516	128	21	2.42
774	256	9	2.33
1290	512	4	2.24
2322	1024	2	2.24
4386	2048	1	2.25
8514	4096	0	2.19

Table 5.1.1: Data on Garbage Collections of Nursery

Our garbage collector was able to achieve performance comparable to the malloc/free version starting with a nursery of size 128 kbytes. As can be seen, the performance improves as we increase the size of the nursery. The garbage collected version was slightly slower compared to the version which did not free any memory.

The second experimental workload repeatedly builds a large linked list and then discards every alternate entry in the list. In this example, the application requested a total of 2064 kbytes and freed 2048 kbytes. The test results for this example are shown in Table 5.1.2. The malloc/free version of this program took 9.22 seconds to execute and used up 63 kbytes of memory. The same application took 2.08 seconds to execute and used 4139 kbytes of memory if the free calls were commented out. The version which partially frees memory, used 2108 kbytes of memory and took 5.34 seconds to execute.

The garbage collected version performed much better compared with the malloc/free version for nursery sizes of at least 256 kbytes. The results with nurseries of smaller sizes have been used to report the performance on complete collections. The performance of the garbage collected version was comparable to the version which did not free any memory.

Total Heap Size (KBytes)	*Size of Nursery (KBytes)*	*Number of Collections of Nursery*	*Total Time (Seconds)*
774	256	42	2.85
1290	512	18	2.29
2322	1024	8	2.10
4386	2048	4	2.10
8514	4096	2	2.10
16770	8192	1	2.11
33282	16384	0	2.09

Table 5.1.2: Data on Garbage Collections of Nursery

The third experimental workload creates a doubly-linked list of objects. It then discards every alternate object and repeats the process a certain number of times. The application requested a total of 1536 kbytes and freed 1512 kbytes. The test results for this example are shown in Table 5.1.3. The malloc/free version of this program took 6.46 seconds to execute and used 75 kbytes of memory. The application took 2.99 seconds to execute and used 3083 kbytes of memory if it did not free any memory. The same application took 4.76 seconds to execute when only a portion of the allocated memory was freed using free. The application in this case used 1584 kbytes of memory.

Total Heap Size (KBytes)	*Size of Nursery (KBytes)*	*Number of Collections of Nursery*	*Total Time (Seconds)*
516	128	43	4.01
774	256	18	3.29
1290	512	8	3.12
2322	1024	4	3.04
4386	2048	2	3.02
8514	4096	0	2.96

Table 5.1.3: Data on Garbage Collections of Nursery

The garbage collected version performed much better compared to the malloc/free version starting with a nursery of size 128 kbytes.

5.2. Complete Garbage Collections

To enable us to collect data on complete collections, the second example described in section 5.1 was executed with old generations and nurseries of different sizes. We have tabulated these results in Table 5.2.1.

Total Heap Size (KBytes)	*Size of the Old Generation (KBytes)*	*Size of the Nursery (KBytes)*	*Number of Collections of Nursery*	*Number of Collections of Old Generation*	*Total Time (Seconds)*
516	256	128	115	2	6.36
548	256	144	89	1	4.73
580	320	128	119	1	6.65
645	384	128	121	1	6.96
774	512	128	127	1	8.04

Table 5.2.1: Data on Complete Collections

The garbage collected version again performed better than the malloc/free version. The performance showed improvement if the old generation was garbage collected less often. As mentioned in the earlier sections, mark-and-sweep techniques sweep through the entire heap. Their performance suffers in systems with large heaps. This observation is also demonstrated by the results shown in Table 5.2.1.

5.2.1. Alternative Techniques for Maintaining Cross References

The method used for maintaining the cross-generational references is a major factor in influencing the performance of the generational technique. Several alternative approaches are possible. Two such techniques are presented here.

1.

a. Organize the old generation as a collection of pages.

b. Write-protect all the old generation pages at the start of the application.

c. Use a page-fault handler to keep track of the old generation pages which have been modified since the last collection. Unprotect a page on a fault.

d. During collection, scan through the modified pages looking for pointers to the nursery. Add the objects holding cross-generational references to the remembered set and reprotect the pages.

2.

a. Organize the old generation as a collection of pages.

b. Write protect all the old generation pages at the start of the application.

c. On a page-fault, the fault-handler adds the modified object to the remembered set.

d. During collection, scan through the remembered set. If an object in the remembered set holds no pointer to the nursery it can be deleted from the remembered set.

Which technique performs best depends on a number of workload characteristics, such as the cost for protecting a page, the number of times the old generation is modified, and the number of cross-generational references.

We have implemented the second technique presented above. For monitoring the writes to the old generation we made use of the fault interpreter utility described in reference 4. The old generation pages are write-protected at the start of an application and all the modified objects in the old generation are logged to the remembered set. Note that this technique causes a page-fault on every write to the old generation (except during the collection process when the old generation pages are not write protected). The cost of handling a page-fault plays a major role in the efficiency of this technique.

The preliminary results obtained on this particular technique have not been very encouraging. The second test case described in the previous section took a total of 23.15 seconds to execute and caused a total of 16161 page faults. We measured the cost of handling 16161 page faults which took a total of 19.60 seconds (We obtained these results on a different machine. The version which does not use page protection technique took 8.94 seconds to execute). The cost of handling the page faults is the dominant factor in this particular implementation.

We believe that the first technique might perform better as it causes at most one page fault per page. The over head in this case will be the cost of scanning the entire modified page. We are yet to implement this technique.

5.3. Conclusions and Future Work

The preliminary results which we have obtained on the performance of our garbage collection technique have been encouraging. We still feel that there is scope for improvement of our technique. Once the compiler support is ready, we plan to test our garbage collector with real applications and fine tune its performance.

As mentioned in our introduction to the generational technique, several tunable parameters exist. It is not obvious whether the mark-and-sweep technique is best for garbage collecting the old generation. It will eventually be necessary to implement a variety of alternative techniques and compare their performance. Based on these comparisons, we will be better able to select values for the tunable parameters.

As mentioned above, generational techniques are known to achieve short pause times during collections. The occasional long pause times observed during complete collections might be unacceptable in some situations. Most of the garbage collection techniques can be made to execute in parallel with actual application execution. Techniques for making the garbage collection process *mostly parallel* have been described in reference 2. We are in the process of implementing one such technique.

6. Acknowledgements

I would like to express my gratitude to Dr. Kelvin Nilsen, my advisor, for his constant encouragement, support, and advice throughout the course of this project. He has spent countless number of hours discussing my work and suggesting new ideas to explore. Working with him has been fun and rewarding.

I would also like to thank Dr. Suresh Kothari and Dr. Charles Wright, my Program of Study Committee members, for their valuable time and co-operation.

Thanks are also due to my family members, especially my parents, who have always given me the necessary support and freedom to explore and realize my plans. Without their support and encouragement, it wouldn't have been possible to accomplish what ever little I have.

Finally, I am grateful to all my friends at Iowa State University for making my stay here fun and memorable.

7. References

1. J. F. Bartlett, A Generational, Compacting Garbage Collector for C++, *Position Paper for OOPSLA/ECOOP Workshop on Garbage Collection in Object-Oriented Systems*, October 1990.

2. H. J. Boehm, A. J. Demers and S. Shenker, Mostly Parallel Garbage Collection, *Proceedings of the ACM SIGPLAN Notices Conference on Programming Language Design and Implementation*, June 1991.

3. P. Caudill, How to Write a Generation Scavenging Collector.

4. D. R. Edelson, Fault Interpretation: Fine-Grain Monitoring of Page Accesses, *Technical Report UCSC-CRL-92-32, University of California at Santa Cruz Computer and Information Science Department*, November 1992.

5. J. R. Ellis and D. L. Detlefs, Safe, Efficient Garbage Collection for C++, Digital Equipment Corporation Systems Research Center Report 102, June 1993.

6. H. Gao and K. Nilsen, Reliable General Purpose Dynamic Memory Management for Real-Time Systems, *IEEE Real-Time Systems Symposium*, submitted.

7. K. Kuse and T. Kamimura, Generational Garbage Collection for C-Based Object-Oriented Languages.

8. K. Nilsen and W. J. Schmidt, Cost-Effective Object-Space Management for Hardware-Assisted Real-Time Garbage Collection, *ACM Letters on Prog. Lang. and Systems 1*, 4 (December 1992), 338-354.

9. K. Nilsen, Reliable Real-Time Garbage Collection of C++, *Computing Systems 7*, 4 (Fall 1994), .

10. K. Nilsen, Cost-Effective Hardware-Assisted Real-Time Garbage Collection, *ACM SIGPLAN Notices Workshop on Language, Compiler, and Tool Support for Real-Time Sy stems*, June 1994.

11. K. D. Nilsen and W. J. Schmidt, A High-Performance Hardware-Assisted Real-Time Garbage Collection System, *Journal of Programming Languages*, To appear.

12. *SPARC International, Inc., The SPARC Architecture Manual*, Published by Prentice Hall, 1992.

13. W. J. Schmidt, Issues in the Design and Implementation of a Real-Time Garbage Collection Architecture, Ph.D. Dissertation, Iowa State Univ. Tech. Rep. 92-25, 1992.

14. P. R. Wilson, M. S. Lam and T. G. Moher, Caching Considerations for Generational Garbage Collection: a Case for Large and Set-Associative Caches, *Technical Report UIC-EECS-90-5, University of Illinois at Chicago EECS Department*, December 1990.

15. P. R. Wilson, Uniprocessor Garbage Collection Techniques, in *Memory Management*, Y. Bekkers and J. Cohen (ed.), Springer-Verlag , 1992, 1-42.